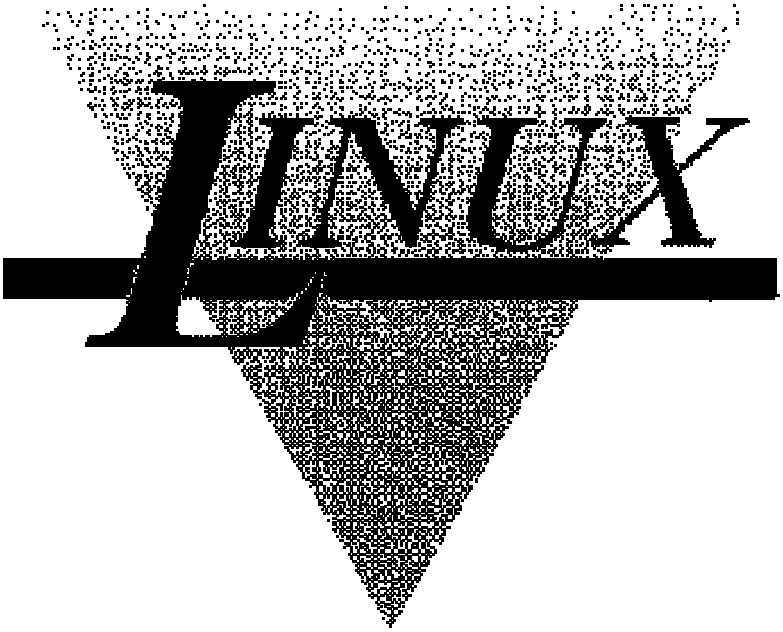
The Linux Programmer's Guide



Sven Goldt
Sven van der Meer
Scott Burkett
Matt Welsh

Version 0.4
March 1995

[0] ...Our continuing mission: to seek out knowledge of C, to explore strange unix commands, and to boldly code where no one has man page 4.

Chapter 6

Linux Interprocess Communications

B. Scott Burkett, scottb@intnet.net v1.0, 29 March 1995

6.1 Introduction

The Linux IPC (Inter-process communication) facilities provide a method for multiple processes to communicate with one another. There are several methods of IPC available to Linux C programmers:

- Half-duplex UNIX pipes
- FIFOs (named pipes)
- SYSV style message queues
- SYSV style semaphore sets
- SYSV style shared memory segments
- Networking sockets (Berkeley style) (not covered in this paper)
- Full-duplex pipes (STREAMS pipes) (not covered in this paper)

These facilities, when used effectively, provide a solid framework for client/server development on any UNIX system (including Linux).

6.2 Half-duplex UNIX Pipes

6.2.1 Basic Concepts

Simply put, a *pipe* is a method of connecting the *standard output* of one process to the *standard input* of another. Pipes are the eldest of the IPC tools, having been around since the earliest incarnations of the UNIX operating system. They provide a method of one-way communications (hence the term half-duplex) between processes.

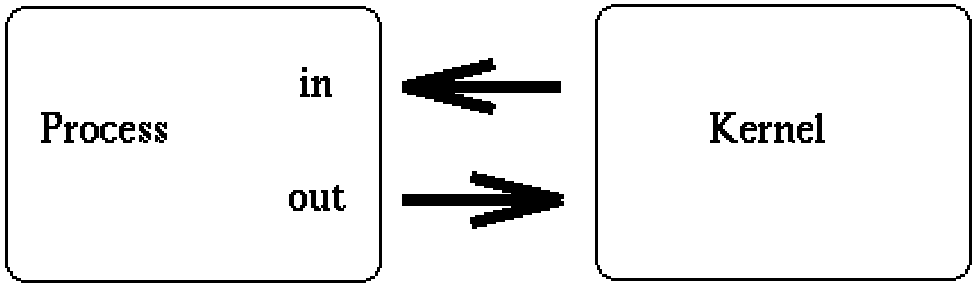
This feature is widely used, even on the UNIX command line (in the shell).

```
ls | sort | lp
```

The above sets up a pipeline, taking the output of ls as the input of sort, and the output of sort as the input of lp. The data is running through a half duplex pipe, traveling (visually) left to right through the pipeline.

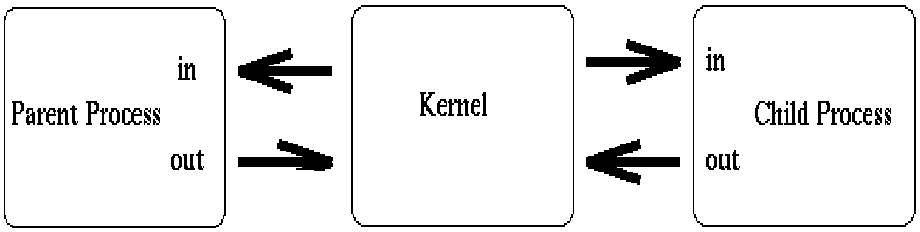
Although most of us use pipes quite religiously in shell script programming, we often do so without giving a second thought to what transpires at the kernel level.

When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe. One descriptor is used to allow a path of input into the pipe (write), while the other is used to obtain data from the pipe (read). At this point, the pipe is of little practical use, as the creating process can only use the pipe to communicate with itself. Consider this representation of a process and the kernel after a pipe has been created:

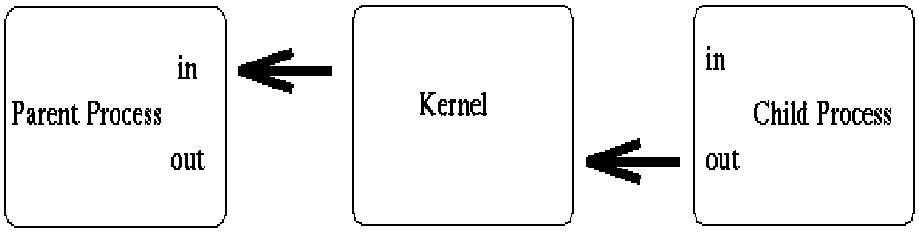


From the above diagram, it is easy to see how the descriptors are connected together. If the process sends data through the pipe (fd0), it has the ability to obtain (read) that information from fd1. However, there is a much larger objective of the simplistic sketch above. While a pipe initially connects a process to itself, data traveling through the pipe moves through the kernel. Under Linux, in particular, pipes are actually represented internally with a valid inode. Of course, this inode resides within the kernel itself, and not within the bounds of any physical file system. This particular point will open up some pretty handy I/O doors for us, as we will see a bit later on.

At this point, the pipe is fairly useless. After all, why go to the trouble of creating a pipe if we are only going to talk to ourself? At this point, the creating process typically forks a child process. Since a child process will inherit any open file descriptors from the parent, we now have the basis for multiprocess communication (between parent and child). Consider this updated version of our simple sketch:



Above, we see that both processes now have access to the file descriptors which constitute the pipeline. It is at this stage, that a critical decision must be made. In which direction do we desire data to travel? Does the child process send information to the parent, or vice-versa? The two processes mutually agree on this issue, and proceed to "close" the end of the pipe that they are not concerned with. For discussion purposes, let's say the child performs some processing, and sends information back through the pipe to the parent. Our newly revised sketch would appear as such:



Construction of the pipeline is now complete! The only thing left to do is make use of the pipe. To access a pipe directly, the same system calls that are used for low-level file I/O can be used (recall that pipes are actually represented internally as a valid inode).

To send data to the pipe, we use the write() system call, and to retrieve data from the pipe, we use the read() system call. Remember, low-level file I/O system calls work with file descriptors! However, keep in mind that certain system calls, such as lseek(), do not work with descriptors to pipes.

6.2.2 Creating Pipes in C

Creating "pipelines" with the C programming language can be a bit more involved than our simple shell example. To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline. After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).

```
SYSTEM CALL: pipe();

PROTOTYPE: int pipe( int fd[2] );
  RETURNS: 0 on success
           -1 on error: errno = EMFILE (no free descriptors)
                                EMFILE (system file table is full)
                                EFAULT (fd array is not valid)

NOTES: fd[0] is set up for reading, fd[1] is set up for writing
```

The first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing. Visually speaking, the output of fd1 becomes the input for fd0. Once again, all data traveling through the pipe moves through the kernel.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
        int     fd[2];

        pipe(fd);
        .
        .
}
```

Remember that an array name in C *decays* into a pointer to its first member. Above, `fd` is equivalent to `&fd[0]`. Once we have established the pipeline, we then fork our new child process:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
}
```

If the parent wants to receive data from the child, it should close fd1, and the child should close fd0. If the parent wants to send data to the child, it should close fd0, and the child should close fd1. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with. On a technical note, the EOF will never be returned if the unnecessary ends of the pipe are not explicitly closed.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);
        }
        else
        {
```

```
                        /* Parent process closes up output side of pipe */
                        close(fd[1]);
                }
                .
                .
        }
```

As mentioned previously, once the pipeline has been established, the file descriptors may be treated like descriptors to normal files.

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, strlen(string));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
```

```
        }

        return(0);
}
```

Often, the descriptors in the child are duplicated onto standard input or output. The child can then exec() another program, which inherits the standard streams. Let's look at the dup() system call:

```
SYSTEM CALL: dup();

PROTOTYPE: int dup( int oldfd );
  RETURNS: new descriptor on success
           -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                EBADF (newfd is out of range)
                                EMFILE (too many descriptors for the process)

NOTES: the old descriptor is not closed!  Both may be used interchangeably
```

Although the old descriptor and the newly created descriptor can be used interchangeably, we will typically close one of the standard streams first. The dup() system call uses the lowest-numbered, unused descriptor for the new one.

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close up standard input of the child */
                close(0);

                /* Duplicate the input side of pipe to stdin */
                dup(fd[0]);
                execlp("sort", "sort", NULL);
                .
        }
```

Since file descriptor 0 (stdin) was closed, the call to dup() duplicated the input descriptor of the pipe (fd0) onto its standard input. We then make a call to execlp(), to overlay the child's text segment (code) with that of the sort program. Since newly exec'd programs inherit standard streams from their spawners, it actually inherits the input side of the pipe as its standard input! Now, anything that the original parent process sends to the pipe, goes into the sort facility.

There is another system call, dup2(), which can be used as well. This particular call originated with Version 7 of UNIX, and was carried on through the BSD releases and is now required by the POSIX standard.

```
SYSTEM CALL: dup2();

PROTOTYPE: int dup2( int oldfd, int newfd );
  RETURNS: new descriptor on success
           -1 on error: errno = EBADF (oldfd is not a valid descriptor)
```

```
                                        EBADF (newfd is out of range)
                                        EMFILE (too many descriptors for the process)

  NOTES: the old descriptor is closed with dup2()!
```

With this particular call, we have the close operation, and the actual descriptor duplication, wrapped up in one system call. In addition, it is guaranteed to be atomic, which essentially means that it will never be interrupted by an arriving signal. The entire operation will transpire before returning control to the kernel for signal dispatching. With the original dup() system call, programmers had to perform a close() operation before calling it. That resulted in two system calls, with a small degree of vulnerability in the brief amount of time which elapsed between them. If a signal arrived during that brief instance, the descriptor duplication would fail. Of course, dup2() solves this problem for us.

Consider:

```
.
.
childpid = fork();

if(childpid == 0)
{
        /* Close stdin, duplicate the input side of pipe to stdin */
        dup2(0, fd[0]);
        execlp("sort", "sort", NULL);
        .
        .
}
```

6.2.3 Pipes the Easy Way!

If all of the above ramblings seem like a very round-about way of creating and utilizing pipes, there is an alternative.

```
  LIBRARY FUNCTION: popen();

  PROTOTYPE: FILE *popen ( char *command, char *type);
    RETURNS: new file stream on success
             NULL on unsuccessful fork() or pipe() call

  NOTES: creates a pipe, and performs fork/exec operations using "command"
```

This standard library function creates a half-duplex pipeline by calling pipe() internally. It then forks a child process, execs the Bourne shell, and executes the ”command” argument within the shell. Direction of data flow is determined by the second argument, ”type”. It can be ”r” or ”w”, for ”read” or ”write”. It cannot be both! Under Linux, the pipe will be opened up in the mode specified by the first character of the ”type” argument. So, if you try to pass ”rw”, it will only open it up in ”read” mode.

While this library function performs quite a bit of the dirty work for you, there is a substantial tradeoff. You lose the fine control you once had by using the pipe() system call, and handling the fork/exec yourself. However, since the Bourne shell is used directly, shell metacharacter expansion (including wildcards) is permissible within the ”command” argument.

Pipes which are created with popen() must be closed with pclose(). By now, you have probably realized that popen/pclose share a striking resemblance to the standard file stream I/O functions fopen() and fclose().

```
  LIBRARY FUNCTION: pclose();

  PROTOTYPE: int pclose( FILE *stream );
    RETURNS: exit status of wait4() call
             -1 if "stream" is not valid, or if wait4() fails

  NOTES: waits on the pipe process to terminate, then closes the stream.
```

The pclose() function performs a wait4() on the process forked by popen(). When it returns, it destroys the pipe and the file stream. Once again, it is synonymous with the fclose() function for normal stream-based file I/O.

Consider this example, which opens up a pipe to the sort command, and proceeds to sort an array of strings:

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen1.c
 *************************************************************************/

#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
        int  cntr;
        FILE *pipe_fp;
        char *strings[MAXSTRS] = { "echo", "bravo", "alpha",
                                   "charlie", "delta"};

        /* Create one way pipe line with call to popen() */
        if (( pipe_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Processing loop */
        for(cntr=0; cntr<MAXSTRS; cntr++) {
                fputs(strings[cntr], pipe_fp);
                fputc('\n', pipe_fp);
        }

        /* Close the pipe */
        pclose(pipe_fp);

        return(0);
}
```

Since `popen()` uses the shell to do its bidding, all shell expansion characters and metacharacters are available for use! In addition, more advanced techniques such as redi-

rection, and even output piping, can be utilized with `popen()`. Consider the following sample calls:

```
popen("ls ~scottb", "r");
popen("sort > /tmp/foo", "w");
popen("sort | uniq | more", "w");
```

As another example of popen(), consider this small program, which opens up two pipes (one to the ls command, the other to sort):

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen2.c
 *************************************************************************/

#include <stdio.h>

int main(void)
{
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];

        /* Create one way pipe line with call to popen() */
        if (( pipein_fp = popen("ls", "r")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Create one way pipe line with call to popen() */
        if (( pipeout_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Processing loop */
        while(fgets(readbuf, 80, pipein_fp))
                fputs(readbuf, pipeout_fp);

        /* Close the pipes */
        pclose(pipein_fp);
        pclose(pipeout_fp);

        return(0);
}
```

For our final demonstration of popen(), let's create a generic program that opens up a pipeline between a passed command and filename:

```
/*************************************************************************
```

```
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 ***************************************************************************
 MODULE: popen3.c
 ***************************************************************************/

#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *pipe_fp, *infile;
        char readbuf[80];

        if( argc != 3) {
                fprintf(stderr, "USAGE:  popen3 [command] [filename]\n");
                exit(1);
        }

        /* Open up input file */
        if (( infile = fopen(argv[2], "rt")) == NULL)
        {
                perror("fopen");
                exit(1);
        }

        /* Create one way pipe line with call to popen() */
        if (( pipe_fp = popen(argv[1], "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Processing loop */
        do {
                fgets(readbuf, 80, infile);
                if(feof(infile)) break;

                fputs(readbuf, pipe_fp);
        } while(!feof(infile));

        fclose(infile);
        pclose(pipe_fp);

        return(0);
}
```

Try this program out, with the following invocations:

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

6.2.4 Atomic Operations with Pipes

In order for an operation to be considered "atomic", it must not be interrupted for any reason at all. The entire operation occurs at once. The POSIX standard dictates in /usr/include/posix1_lim.h that the maximum buffer size for an atomic operation on a pipe is:

```
#define _POSIX_PIPE_BUF          512
```

Up to 512 bytes can be written or retrieved from a pipe atomically. Anything that crosses this threshold will be split, and not atomic. Under Linux, however, the atomic operational limit is defined in "linux/limits.h" as:

```
#define PIPE_BUF         4096
```

As you can see, Linux accommodates the minimum number of bytes required by POSIX, quite considerably I might add. The atomicity of a pipe operation becomes important when more than one process is involved (FIFOS). For example, if the number of bytes written to a pipe exceeds the atomic limit for a single operation, and multiple processes are writing to the pipe, the data will be "interleaved" or "chunked". In other words, one process may insert data into the pipeline between the writes of another.

6.2.5 Notes on half-duplex pipes:

- Two way pipes can be created by opening up two pipes, and properly reassigning the file descriptors in the child process.
- The pipe() call must be made BEFORE a call to fork(), or the descriptors will not be inherited by the child! (same for popen()).
- With half-duplex pipes, any connected processes must share a related ancestry. Since the pipe resides within the confines of the kernel, any process that is not in the ancestry for the creator of the pipe has no way of addressing it. This is not the case with named pipes (FIFOS).

6.3 Named Pipes (FIFOs - First In First Out)

6.3.1 Basic Concepts

A named pipe works much like a regular pipe, but does have some noticeable differences.

- Named pipes exist as a device special file in the file system.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

6.3.2 Creating a FIFO

There are several ways of creating a named pipe. The first two can be done directly from the shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

The above two commands perform identical operations, with one exception. The mkfifo command provides a hook for altering the permissions on the FIFO file directly after creation. With mknod, a quick call to the chmod command will be necessary.

FIFO files can be quickly identified in a physical file system by the "p" indicator seen here in a long directory listing:

```
$ ls -l MYFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MYFIFO|
```

Also notice the vertical bar ("pipe sign") located directly after the file name. Another great reason to run Linux, eh?

To create a FIFO in C, we can make use of the mknod() system call:

```
LIBRARY FUNCTION: mknod();

PROTOTYPE: int mknod( char *pathname, mode_t mode, dev_t dev);
  RETURNS: 0 on success,
           -1 on error: errno = EFAULT (pathname invalid)
                                EACCES (permission denied)
                                ENAMETOOLONG (pathname too long)
                                ENOENT (invalid pathname)
                                ENOTDIR (invalid pathname)
                                (see man page for mknod for others)

NOTES: Creates a filesystem node (file, device file, or FIFO)
```

I will leave a more detailed discussion of mknod() to the man page, but let's consider a simple example of FIFO creation from C:

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In this case, the file "/tmp/MYFIFO" is created as a FIFO file. The requested permissions are "0666", although they are affected by the umask setting as follows:

```
final_umask = requested_permissions & ~original_umask
```

A common trick is to use the umask() system call to temporarily zap the umask value:

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In addition, the third argument to mknod() is ignored unless we are creating a device file. In that instance, it should specify the major and minor numbers of the device file.

6.3.3 FIFO Operations

I/O operations on a FIFO are essentially the same as for normal pipes, with once major exception. An "open" system call or library function should be used to physically open up a channel to the pipe. With half-duplex pipes, this is unnecessary, since the pipe resides in the kernel and not on a physical filesystem. In our examples, we will treat the pipe as a stream, opening it up with fopen(), and closing it with fclose().

Consider a simple server process:

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: fifoserver.c
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
        }

        return(0);
}
```

Since a FIFO blocks by default, run the server in the background after you compile it:

```
$ fifoserver&
```

We will discuss a FIFO's blocking action in a moment. First, consider the following simple client frontend to our server:

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: fifoclient.c
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE       "MYFIFO"
```

```
int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

6.3.4 Blocking Actions on a FIFO

Normally, blocking occurs on a FIFO. In other words, if the FIFO is opened for reading, the process will ”block” until some other process opens it for writing. This action works vice-versa as well. If this behavior is undesirable, the O_NONBLOCK flag can be used in an open() call to disable the default blocking action.

In the case with our simple server, we just shoved it into the background, and let it do its blocking there. The alternative would be to jump to another virtual console and run the client end, switching back and forth to see the resulting action.

6.3.5 The Infamous SIGPIPE Signal

On a last note, pipes must have a reader and a writer. If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline.

6.4 System V IPC

6.4.1 Fundamental Concepts

With System V, AT&T introduced three new forms of IPC facilities (message queues, semaphores, and shared memory). While the POSIX committee has not yet completed its standardization of these facilities, most implementations do support these. In addition, Berkeley (BSD) uses sockets as its primary form of IPC, rather than the System V elements. Linux has the ability to use both forms of IPC (BSD and System V), although we will not discuss sockets until a later chapter.

The Linux implementation of System V IPC was authored by *Krishna Balasubramanian*, at `balasub@cis.ohio-state.edu`.

IPC Identifiers

Each IPC *object* has a unique IPC identifier associated with it. When we say “IPC object”, we are speaking of a single message queue, semaphore set, or shared memory segment.